MEMORY STACKS IN 8086 MICROPROCESSORS

STACK

The stack is a block of memory that may be used for temporarily storing the contents of the registers inside the CPU. It is a top-down data structure whose elements are accessed using the stack pointer (SP) which gets decremented by two as we store a data word into the stack and gets incremented by two as we retrieve a data word from the stack back to the CPU register. The process of storing the data in the stack is called '**pushing into**' the stack and the

reverse process of transferring the data back from the stack to the CPU register is known as

'**popping off**' the stack. The stack is essentially *Last-In-First-Out* (LIFO) data segment. This

means that the data which is pushed into the stack last will be on top of stack and will be popped off the stack first.

STACK STRUCTURE OF 8086

Stack contains a set of sequentially arranged data types, with the last item appearing on top of the stack. This item will be popped off the stack first for use by the CPU. The stack pointer is a 16-bit register that contains the offset address of the memory location in the stack segment. The stack segment, like any other segment, may have a memory block of a maximum of 64 Kbytes locations, and thus may overlap with any other segments. Stack Segment register (SS) contains the base address of the stack segment in the memory.

The Stack Segment register (SS) and Stack pointer register (SP) together address the stack-top as explained below:

SS ⟹ 5000H
SP ⟹ 2050H

Let the content of SS be 5000H and the content of the stack pointer register be 2050H.To find the current stack top address ,the stack segment register content is shifted left by four bit positions (multiplied by 10H) and the resulting 20 bit content is added with 16 bit offset value ,stored in the stack pointer register.

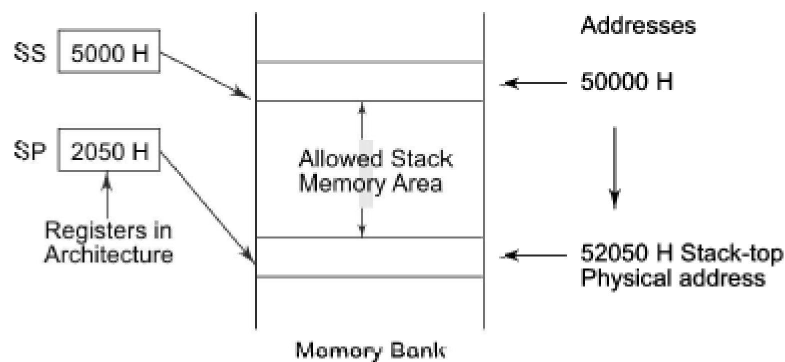
SS	⇒ 5000 H					
SP	⇒ 2050 H					
SS	⇒	0101	0000	0000	0000	
10H * SS	⇒	0101	0000	0000	0000	0000
	+					
SP	⇒		0010	0000	0101	0000
Stack-top		0101	0010	0000	0101	0000
address		5	2	0	5	0

If the stack top points to a memory location 52050H, it means that the location 52050H is already occupied with the previously pushed data. The next 16 bit push operation will

decrement the stack pointer by two, so that it will point to the new stack-top 5204EH and the decremented contents of SP will be 204EH. This location will now be occupied by the recently pushed data.

Thus for a selected value of SS, the maximum value of SP=FFFFH and the segment can have maximum of 64K locations. If the SP starts with an initial value of FFFFH, it will be decremented by two whenever a 16-bit data is pushed onto the stack. After successive push operations, when the stack pointer contains 0000H, any attempt to further push the data to the stack will result in stack overflow.

After a procedure is called using the CALL instruction, the IP is incremented to the next instruction. Then the contents of IP, CS and flag register are pushed automatically to the stack. The control is then transferred to the specified address in the CALL instruction i.e. starting address of the procedure. Then the procedure is executed.



If the stack top points to a memory location 52050 H it means that the location 52050H is already occupied ie, Previously pushed data is available at 52050 H. The next 16-bit push operation will decrement the stack pointer by two, so that it will point to the new stack top 5204E H, and the decremented contents of SP will be 204E H. This location will now be occupied by the recently pushed data. Thus if a 16-bit data is pushed onto the stack ,the push operation will decrement the SP by two because two locations will be required for a 2-byte data

The maximum value of SP =FFFF H and segment can have maximum of 64K locations. Thus after starting with an initial vale of FFFF the stack pointer is decremented by two whenever a 16 bit data is pushed into the stack. After successive push operation when the stack pointer contains 0000 H any attempt to further push the data to the stack will result in stack overflow

Each PUSH operation decrements SP by two, while each POP operation increments the SP.The POP operation is used to retrieve the data stored on to the stack. Fig 4.2 shows the stack overflow conditions while Fig 4.3 shows the effect of push and pop operations on the stack memory block.

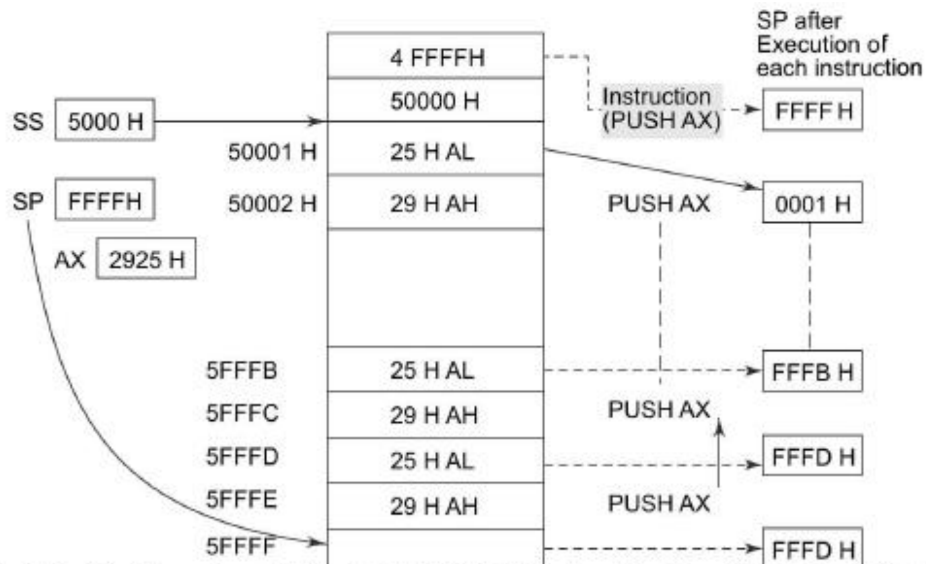


Fig. 4.2 *The Execution of Bracketed PUSH AX Instruction Results in Stack Overflow*

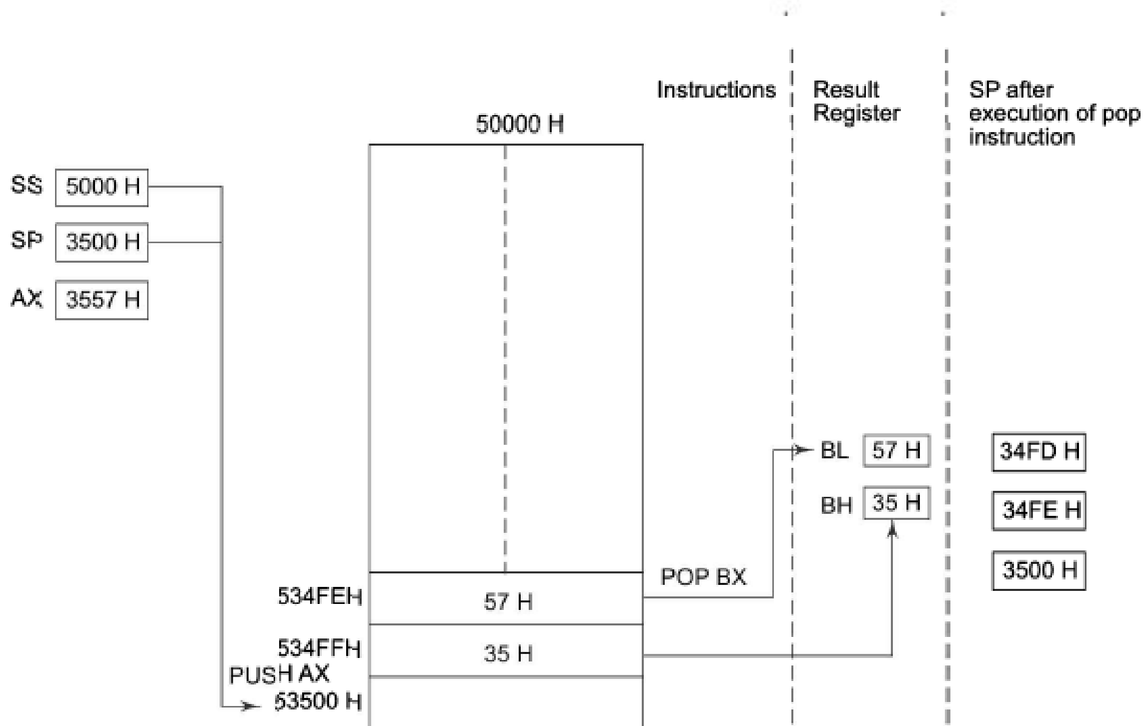


Fig. 4.3 *Effect of PUSH and POP on SP*

Programming using stack

The 8086 has four segement registers namely CS ,DS,SS,and ES.Out of these segments SS contains the segment value for stack while SP contains the offset for the stack -top Ina program the stack segment can be defined in a similar way as the data segment .The ASSUME directives directs the name of stack segment to assembler.

Write a program to calculate squares of BCD numbers 0to 9 and store them sequentially from 2000H offsets onwards in the current data segment. Write a subroutine for the calculation of the square of a number

```
ASSUME CS : CODE, DS : DATA, SS : STACK
DATA     SEGMENT
         ORG 2000H
SQUARES DB OFH DUP (?)
DATA     ENDS
STACK    SEGMENT
STAKDATA DB 100H DUP (?)          ; Reserve 256 bytes for stack
STACK    ENDS
CODE     SEGMENT
START:  MOV AX, DATA              ; Initialise data segment
        MOV DS, AX
        MOV AX, STACK             ; Initialise stack segment
        MOV SS, AX
        MOV SP, OFFSET STAKDATA   ; Initialise stack pointer
        MOV CL, OAH               ; Initialise counter for numbers
        MOV SI, OFFSET SQUARES    ; Set pointer to array of squares
```

```
        MOV AL, 00                ; Start from 00
NEXTNUM : CALL SQUARE             ; Calculate square
        MOV BYTE PTR [SI],AH      ; Store square in the array
        INC AL                    ; Go for the next number
        INC SI                    ; Increment the array pointer
        DCR CL                    ; Decrement count
        JNZ NEXTNUM               ; Stop, if CL = 0, else, continue
        MOV AH, 4CH               ; Return to DOS prompt
        INT 21H                   ;
PROCEDURE SQUARE NEAR             ; SQUARE is a local procedure
                                  ; called only by this segment
        MOV BH,AL                 ;
        MOV CH,AL                 ; Duplicate AL to CH and BH
        XOR AL,AL                 ; Clear flags and AL
AGAIN : ADD AL,CH                 ; Successively add CH to AH
        DAA                       ; Get BCD equivalent
        DCR CH                    ; Decrement successive addition
        JNZ AGAIN                 ; counter till it becomes
        MOV AH,AL                 ; Zero, store the square and
        MOV AL,BH                 ; get back the original number
        RET                       ; Return
SQUARE      ENDP
CODE  ENDS
    END START
```

INTERRUPTS IN 8086

While the CPU is executing a program, an interrupt breaks the normal execution of instructions, diverts its execution to some other program called Interrupt Service Routine (ISR) Whenever an interrupt occurs the processor completes the execution of the current instruction and starts the execution of an Interrupt Service Routine (ISR) or Interrupt Handler.

ISR is a program that tells the processor what to do when the interrupt occurs. At the end of the ISR the last instruction should be IRET. After the execution of ISR, control returns back to the main routine where it was interrupted.

- Whenever a number of devices interrupt a CPU at a time, and if the processor is able to handle them properly, it is said to have multiple interrupt processing capability.
- There are two interrupt pins in 8086. NMI and INTR

Need for Interrupt: Interrupts are particularly useful when interfacing I/O devices that provide or require data at relatively low data transfer rate.

NMI

It is a single non-maskable interrupt pin (NMI) having higher priority. When this interrupt is activated,

these actions take place −

- Completes the current instruction that is in progress.
- Pushes the Flag register values on to the stack.
- Pushes the CS (code segment) value and IP (instruction pointer) value of the return address on to
- the stack.
- IP is loaded from the contents of the word location 00008H.(Type 2*4=00008 H)
- CS is loaded from the contents of the next word location 0000AH.
- Interrupt flag and trap flag are reset to 0

INTR

The INTR is a maskable interrupt pin. It can be accepted (enable) or rejected (masked).

The microprocessor enabled the interrupt using set interrupt flag instruction. It should disable using clear interrupt Flag instruction.

These actions are taken by the microprocessor −

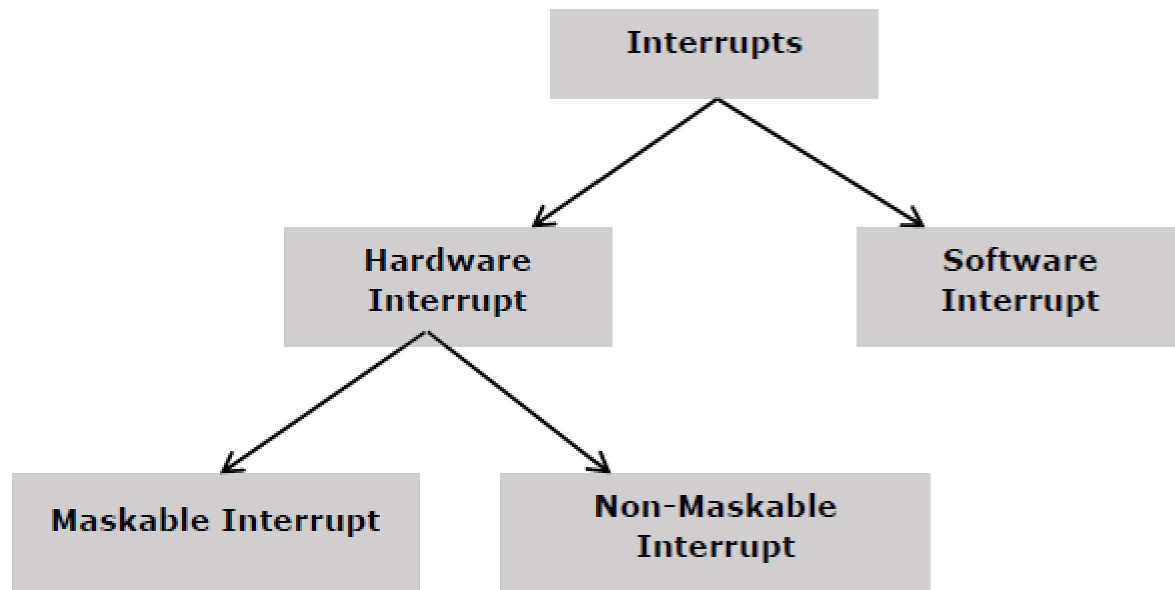
- First completes the current instruction.
- Activates INTA output and receives the interrupt type, say X.
- Flag register value, CS value of the return address and IP value of the return address are pushed
- on to the stack.
- IP value is loaded from the contents of word location $X \times 4$
- CS is loaded from the contents of the next word location.
- Interrupt flag and trap flag is reset to 0

TYPES OF INTERRUPTS

In general there are two types of Interrupts:

Internal (or) Software Interrupts are generated by a software instruction and operate similarly to a jump or branch instruction.

External (or) Hardware Interrupts are caused by an external hardware module.



HARDWARE INTERRUPTS

Hardware interrupts are generated by hardware devices when something unusual happens; this could be a key-press or a mouse move or any other action.

It can be divided into two

1. Maskable 2. Non maskable

- **Maskable Interrupts**:There are some interrupts which can be masked (disabled)or enabled by the processor.
- **Non-Maskable Interrupts**:There are some interrupts which cannot be masked out or ignored by the processor. These are associated with high priority tasks which cannot be ignored (like memory parity or bus faults).

SOFTWARE INTERRUPTS

Interrupts are generated by a software instruction and operate similarly to a jump or branch instruction.

- 256 interrupts : INT n is invoked as software interrupts- n is the type no in the range 0 to 255(00 to FF)

Type 0 to Type4 (Dedicated Interrupts or Predefined Interrupts)

- **TYPE 0** interrupt represents division by zero situation.

- **TYPE 1** interrupt represents single-step execution during the debugging of a program.

- **TYPE 2** interrupt represents non-maskable NMI interrupt.

- **TYPE 3** interrupt represents break-point interrupt.

- **TYPE 4** interrupt represents overflow interrupt.

Type 5 to 31(Not used by 8086, reserved for higher processor like 80286,80386⋯.

Type 32-255(Available for user) : User defined interrupts

Devide by zero(type-0) Interrupt

- This interrupt occurs whenever there is division error i.e. when the result of a division is too large to be stored. This condition normally occurs when the divisor is very small as compared to the dividend or the divisor is zero.
- Its ISR address is stored at location 0 x 4 = 00000H in the IVT.

Single Step(type-1) Interrupt

- The microprocessor executes this interrupt after every instruction if the TF is set.
- It puts microprocessor in single stepping mode i.e. the microprocessor pauses after executing every instruction. This is very useful during debugging.
- Its ISR generally displays contents of all registers. Its ISR address is stored at location 1 x 4 = 00004H in the IVT.

Non-Maskable(type-2) Interrupt

- The microprocessor executes this ISR in response to an interrupt on the NMI (Non mask-able Interrupt) line.
- Its ISR address is stored at location 2 x 4 = 00008H in the IVT.

Breakpoint(type-3) Interrupt

- This interrupt is used to cause breakpoints in the program. It is caused by writing the instruction INT 03H or simply INT.
- It is useful in debugging large programs where single stepping is efficient.
- Its ISR is used to display the contents of all registers on the screen. Its ISR address is stored at location 3 x 4 = 0000CH in the IVT

Overflow (type-4) Interrupt

- This interrupt occurs if the overflow flag is set and the microprocessor executes the INTO (Interrupt on Overflow) instruction.

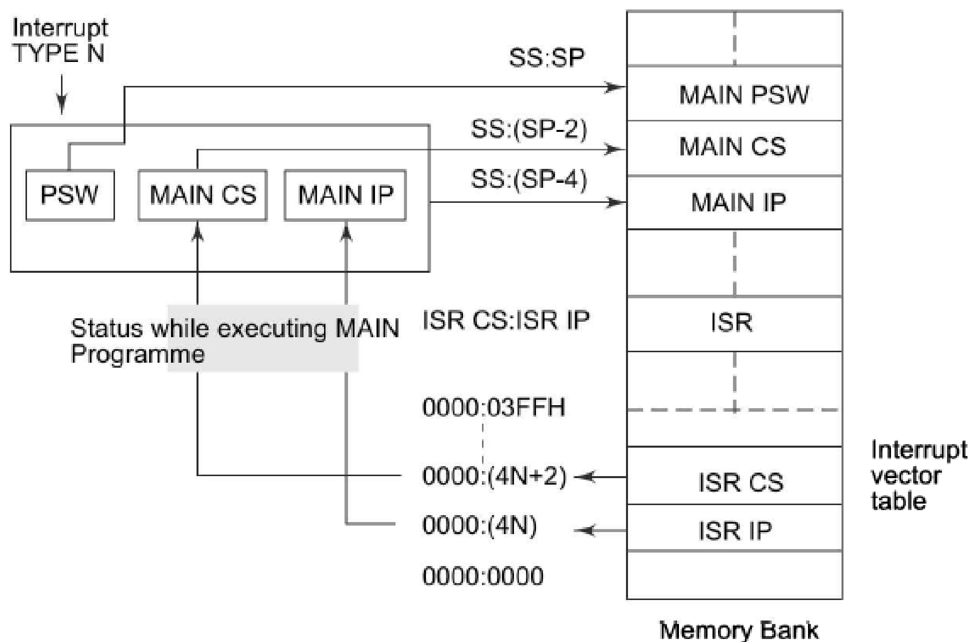
- It is used to detect overflow error in signed arithmetic operations.
- Its ISR address is stored at location 4 x 4 = 00010H in the IVT.

Interrupt Priority

The table shows interrupt priority in 8086

Interrupt	Priority
Divide error, INT n, INTO	Highest
NMI	↓
INTR	↓
Single Step	*Lowest*

Suppose an external device interrupts the CPU at the interrupt pin either NMI or INTR of the 8086,while executing an instruction of a program. The CPU first completes the execution of the current instruction. The IP is then incremented to point to the next instruction .The CPU acknowledges the requesting device on its INTA pin. If it is an INT request ,the CPU checks the IF flag. If the IF is set ,the interrupt request is acknowledged using INTA pin .If the IF is not set, the interrupt requests are ignored.



Interrupt Response Sequence

INTERRUPT SERVICE ROUTINE

For every interrupt, there must be an interrupt service routine (ISR), or interrupt handler. When an interrupt is invoked, the microprocessor runs the interrupt service routine. For every interrupt, there is a fixed location in memory that holds the address of its ISR. The group of memory locations set aside to hold the addresses of ISRs is called the interrupt vector table.

When an interrupt is occurred, the microprocessor stops execution of current instruction. It transfers the content of program counter (CS and IP) into stack.

Interrupt Type	Content (16-bit)	Address	Comments
Type 0	ISR IP	0000:0000	Reserved for divide by Zero interrupt
	ISR CS	0000:0002	
Type 1	ISR IP	0000:0004	Reserved for single step interrupt
	ISR CS	0000:0006	
Type 2	ISR IP	0000:0008	Reserved for NMI
	ISR CS	0000:000A	
Type 3	ISR IP	0000:000C	Reserved for INT single byte instruction
	ISR CS	0000:000E	
Type 4	ISR IP	0000:0010	Reserved for INTO instruction
	ISR CS	0000:0012	
		0000:0014	Reserved for two byte instruction INT TYPE
		0000:0016	
Type N	ISR IP	0000:004N	
	ISR CS	0000:(004N+2)	
		0000:03FC	
Type FFH	ISR IP	0000:03FE	
	ISR CS	0000:03FF	

ISR: Interrupt Service Routine

Structure of Interrupt Vector Table of 8086/88

The first 1Kbyte of memory of 8086 (00000 to003FF) is set aside as a table for storing the starting addresses of Interrupt Service Procedures (ISP). Since 4-bytes are required for storing starting addresses of ISPs, the table can hold 256 Interrupt procedures.

The starting address of an ISP is often called the Interrupt Vector or Interrupt Pointer. Therefore the table is referred as Interrupt Vector Table. Each interrupt type is given a number between

0 to 255 and the address of each interrupt is found by multiplying the type by 4 e.g. for type 11, interrupt address is 11 x 4 = 4410= 0002CH

When the 8086 responds to an interrupt, it automatically goes to the specified location in the Interrupt Vector Table in 8086 to get the starting address of interrupt service routine

<u>Non mask-able interrupt</u>

- 8086 has a non maskable interrupt input pin (NMI) that has the highest priority
- TRAP(Single step TYPE-1) is an internal interrupts having the highest priority among all the interrupts except divide by zero
- On receiving an interrupt on NMI line, the microprocessor executes INT
- Microprocessor obtains the ISR address from location 2 x 4 = 00008H from the IVT.
- It reads 4 locations starting from this address to get the values for IP and CS to execute the ISR.

<u>mask-able interrupt</u>

This is a mask-able, level triggered, low priority interrupt.

- 8086 also provide a pin INTR that has lowest priority as compared to NMI
- On receiving an interrupt on INTR line, the microprocessor executes 2 INTA pulses.
- 1st INTA pulse − The interrupting device calculates (prepares to send) the vector number.
- 2nd INTA pulse − The interrupting device sends the vector number ‘N’ to the microprocessor.
- Now microprocessor multiplies N x 4 and goes to the corresponding location in the IVT to obtain the ISR address. INTR is a mask-able interrupt.
- It is masked by making IF = 0 by software through CLI instruction.
- It is unmasked by making IF = 1 by software through STI instruction.

<u>Interrupt acknowledgment bus cycle</u>

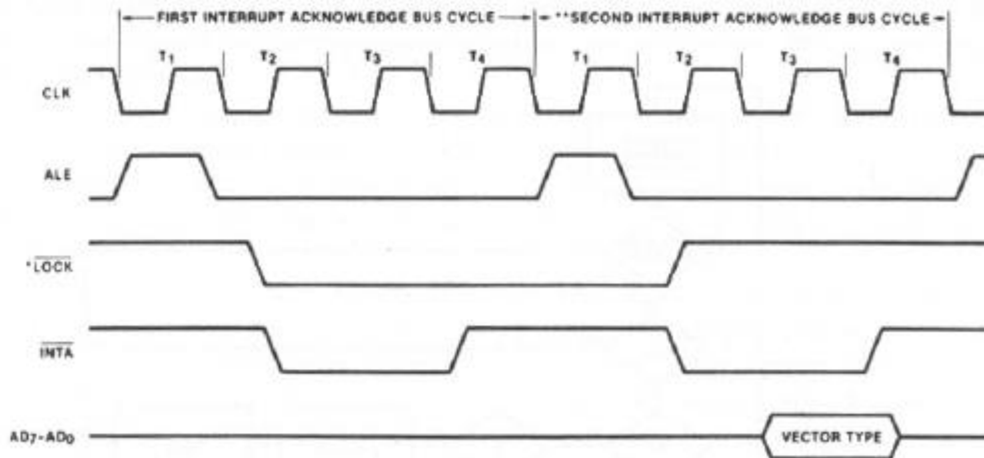


Figure 11–9 Interrupt-acknowledge bus cycle. (Reprinted by permission of Intel Corporation. Copyright/Intel Corp. 1979)

Suppose an external signal interrupts the processor and the pin LOCK goes low at the trailing edge of the first ALE pulse that appears after the interrupt signal preventing the use of bus for any other purpose.

The pin $\overline{LOCK}$ remains low till the start of the next machine cycle.

With the trailing edge of $\overline{LOCK}$, the $\overline{INTA}$ goes low and remains low for two clock states before returning back to the high state.

It remains high till the start of the next machine cycle, i.e. next trailing edge of ALE.

Then $\overline{INTA}$ again goes low, remains low for two states before returning to the high state.The first trailing edge of ALE floats the bus AD_0-AD_7 , while the second trailing edge prepares the bus to accept the type of the interrupt. The type of the interrupt remains on the bus for a period of two cycles.

<u>The Operation of an Interrupt sequence on the 8086 Microprocessor:</u>

1. The SP is decremented by two and the contents of the flag register is pushed to the stack memory
2. The interrupt system is disabled by clearing the interrupt flag(IF)
3. The single step trap flag is disabled by clearing the Trap Flag(TP)
4. The stack pointer is decremented by two and contents of the CS register is pushed to the stack memory
5. Again, stack pointer is decremented by two and contents of the IP register is pushed to the stack memory

6. In case of hardware interrupts through INTR the processor runs an interrupt acknowledge cycle to get the interrupt type number. For software interrupt the type number is specified in the instruction itself
7. The processor generate a 20 bit memory address by multiplying the type number by four and sign extending 20 bit. This memory address is the address of the interrupt vector table
8. The first word pointed by the vector table address is loaded in the IP and next is loaded in the CS register
9. The 20 bit physical address of the ISR is calculated by multiplying the contents of the CS register by 16 and adding it to IP
10. The processor executes the ISR to service the interrupt
11. The ISR will be terminated by IRET instruction. When the instruction is executed ,the top of the stack is popped to the IP,CS and flag register one word by one word
12. Thus at the end of the ISR ,the previous status of the processor is restored and so the processor will resume execution of normal program from the instruction where it was suspended

Interrupt Programming

While programming for any type of interrupt, the programmer must either externally or through the program, set the interrupt vector table for that type preferably with the SC and IP addresses of the interrupt service routine .The method of defining the interrupt service routine for software as well as hardware interrupt is the same Fig 4.7 shows the execution sequences in case of a software interrupt .Fig 4.8 shows the transfer of control for the nested interrupts.

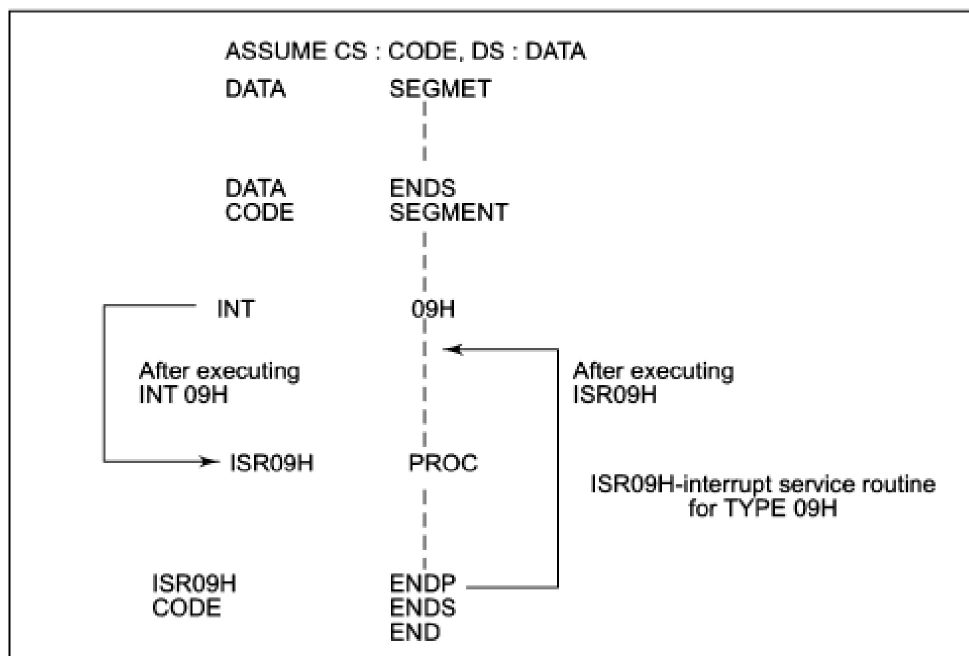


Fig. 4.7 *Transfer of Control during Execution of an Interrupt Service Routine*

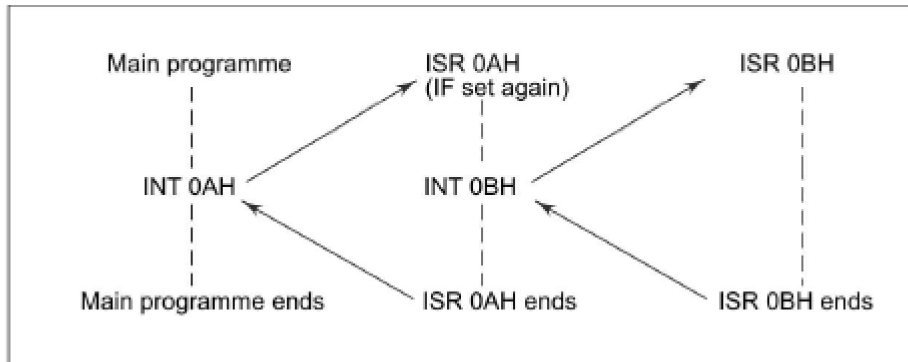


Fig. 4.8 *Transfer of Control for Nested Interrupts*

4.8 PASSING PARAMETERS TO PROCEDURES

Procedures or subroutines may require input data or constants for their execution. Their data or constants may be passed to the subroutine by the main program (host or calling program) or some subroutine may access readily available data of constants available in memory.

Generally, the following techinques are used to pass input data/parameter to procedures in assembly language programs.

(i) Using global declared variable
(ii) Using registers of CPU architecutre
(iii) Using memory locations (reserved)
(iv) Using stack
(v) Using PUBLIC & EXTRN.

Besides these methods if a procedure is interactive it may directly accept inputs from input devices.

As discussed in Chapter 2, a variable or a parameter label may be declared global in the main program and the same variable or parameter label can be used by all the routines or procedures of the application. Examples of passing parameters.

Example 4.1

```
ASSUME CS:CODE1,DS:DATA
DATA SEGMENT
NUMBER EQU 77H GLOBAL
DATA ENDS
CODE1 SEGMENT
    START :             MOV AX,DATA



                   MOV DS,AX
                        •
                        •
                   MOV AX,NUMBER
                        •
                   CODE1 ENDS
 ASSUME CS:CODE2
 CODE2 SEGMENT
            MOV AX,DATA
            MOV DS,AX
            MOV BX,NUMBER

      CODE2  ENDS
            END START
```

The CPU general purpose registers may be used to pass parameters to the procedures. The main program may store the parameters to be passed to the procedure in the available CPU registers and the procedure may use the same register contents for execution. The original contents of the used CPU register may change during execution of the procedure. This may be avoided by pushing all the register content to be used to the stack sequentially at the start of the procedure and by poping all the register contents at the end of the procedure in opposite sequence.

Example 4.2

```
ASSUME CS:CODE
CODE SEGMENT
    START :      MOV AX,5555H
                 MOV BX,7272H
                      •
                      •
                 CALL PROCEDURE1
                      •
                      •
                      •
    PROCEDURE    PROCEDURE1 NEAR
                      •
                      •
                 ADD AX,BX
                      •
                      •
                 RET
    PROCEDURE1   ENDP
    CODE ENDS
                 END START
```

Memory locations may also be used to pass parameters to a procedure in the same way as registers. A main program may store the parameter to be passed to a procedure at a known memory address location and the procedure may use the same location for accessing the parameter.

Example 4.3

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
NUM DB (55H)
COUNT EQU 10H
DATA ENDS
CODE SEGMENT
    START :      MOV AX,DATA
                 MOV DS,AX
                      •
                      •
                 CALL ROUTINE
                      •
                      •
                      •
    PROCEDURE    ROUTINE NEAR
                 MOV BX,NUM
                 MOV CX,COUTN
                      •
    ROUTINE      ENDP
    CODE         ENDS
                 END START
```

Stack memory can also be used to pass parameters to a procedure. A main program may store the parameters to be passed to a procedure in its CPU registers. The registers will further be pushed on to the stack. The procedure during its execution pops back the appropriate parameters as and when required. This procedure of poping back the parameters must be implemented carefully because besides the parameters to be passed to the procedure the stack contains other important information like contents of other pushed registers, return addresses from the current procedure and other procedure or interrupt service routines.

Example 4.4

```
ASSUME CS:CODE, SS:STACK
CODE SEGMENT
    START :      MOV AX,STACK
                 MOV SS,AX
                 MOV AX,5577H
                 MOV BX,2929H
                      •
                 PUSH AX
                 PUSH BX
                 CALL ROUTINE ; Decrements SP by 2 (by 4 far routine)
                      •
                      •
```

```
PROCEDURE    ROUTINE NEAR
                     •
        •
        MOV DX,SP
        ADD SP,02    ;      Leave initial two stack bytes of
                            return offset and
                            segment address after executing
                            subroutine
                            POP BX              ;      The data is
                            POP AX              ;      Passes in BX,AX
                            MOV SP,DX
                                •
                                •
                                •
STACK SEGMENT
STACKDATA DB 200H DUP (?)
STACK ENDS
```

For passing the parameters to procedures using the PUBLIC & EXTRN directives, must be declared PUBLIC (for all routines) in the main routine and the same should be declared EXT RN in the procedure. Thus the main program can pass the PUBLIC parameter to a procedure in which it is declared EXTRN (external)

Example 4.5

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
PUBLIC NUMBER EQU 200H
         DATA ENDS
         CODE SEGMENT
    START :   MOV AX,DATA
              MOV DS,AX
                •
                •
                •
              CALL ROUTINE
                •
                •
                •
              PROCEDURE ROUTINE NEAR
          EXTRN NUMBER
              MOV AX,NUMBER
                •
                •
                •
          ROUTINE   ENDP
```

MACROS

A macro is a named block of assembly language statements. Once defined, it can be invoked (called) as many times in a program as you wish. When you invoke a macro, a copy of its code is inserted directly into the program at the location where it was invoked. This type of automatic

code insertion is also known as inline expansion. It is customary to refer to calling a macro, although technically there is no CALL instruction involved.

Defining a Macro

A macro is defined using the MACRO and ENDM directives

```
macroname MACRO parameter-1, parameter-2...statement-list
.
.
.
ENDM
```

The below definition of a macro assign the name DISPLAY to the instruction sequence between the directives MACRO and ENDM

```
DISPLAY MACRO
        MOV AX, SEG MSG
        MOV DS, AX
        MOV DX, OFFSET MSG
        MOV AH, 09 H
        INT 21 H
ENDM
```

Passing parameters to a MACRO

Using parameters in a definition ,the programmer specifies the parameters of the macro those are likely to be changed each time the macro is called.

```
DISPLAY MACRO MSG
        MOV AX, SEG MSG
        MOV DS, AX
        MOV DX, OFFSET MSG
        MOV AH, 09 H
        INT 21 H
  ENDM
```

This parameter MSG can be replaced by MSG1 or MSG2 while calling the macro as shown

```
                          :
                          :
                    DISPLAY MSG1
                          :
                          :
                    DISPLAY MSG2
                          :
                          :
MSG1 DB 0AH,0DH, "Program Terminated Normally",0AH,0DH, "$"
MSG2 DB 0AH,0DH, "Retry, Abort, Fail",0AH,0DH, "$"
```

Invoking Macros

A macro is called (invoked) by inserting its name in the program, possibly followed by macro arguments.

The syntax for calling a macro is:

macroname argument-1, argument-2, ...

Macroname must be the name of a macro defined prior to this point in the source code. Each argument is a text value that replaces a parameter in the macro. The order of arguments must correspond to the order of parameters, but the number of arguments does not have to match the number of parameters. If too many arguments are passed, the assembler issues a warning. If too few arguments are passed to a macro, the unfilled parameters are left blank.

Procedures

Defining a Procedure

Informally, we can define a procedure as a named block of statements that ends in a return statement. A procedure is declared using the PROC and ENDP directives. It must be assigned a name (a valid identifier). A procedure must be written in the end of the code segment (before end directive), and it can't receive any parameter.

When we create a procedure other than your program's startup procedure, end it with a RET instruction. RET forces the CPU to return to the location from where the procedure was called. A procedure is defined using the PROC and ENDP directives.

The syntax is:

```
procedurename PROC
.
.
Ret
procedurename ENDP
```

Example :SumOf PROC

```
add ax,bx
```

```
add ax,cx
ret
SumOf ENDP
```

Invoking procedures

A procedure is called (invoked) by inserting its name in the program Preceded by (call).

The syntax for calling a procedure is:

```
Call procedurename
```

Difference between Macro and Procedure :

S.No.	MACRO	PROCEDURE
01.	Macro definition contains a set of instruction to support modular programming.	Procedure contains a set of instructions which can be called repetitively which can perform a specific task.
02.	It is used for small set of instructions mostly less than ten instructions.	It is used for large set of instructions mostly more than ten instructions.
03.	In case of macro memory requirement is high.	In case of procedure memory requirement is less.
04.	CALL and RET instruction/statements are not required in macro.	CALL and RET instruction/statements are required in procedure.
05.	Assembler directive MACRO is used to define macro and assembler directive ENDM is used to indicate the body is over.	Assembler directive PROC is used to define procedure and assembler directive ENDP is used to indicate the body is over.
06.	Execution time of macro is less than it executes faster than procedure.	Execution time of procedures is high as it executes slower than macro.
07.	Here machine code is created multiple times as each time machine code is generated when macro is called.	Here machine code is created only once, it is generated only once when the procedure is defined.
08.	In a macro parameter is passed as part of statement that calls macro.	In a procedure parameters are passed in registers and memory locations of stack.
09.	Overhead time does not take place as there is no calling and returning.	Overhead time takes place during calling procedure and returning control to calling program.